A

ASSISTANT COMMISSIONER OF PATENTS
Washington, D.C. 20231

DOCKET NUMBER: UK998092

Sir:

Transmitted herewith for filing is the Patent Application of:

Inventor(s): Martin John Trotter

For: MEMORY MANAGEMENT

Enclosed are:

X Patent Specification and Declaration

X 5 sheets of drawing(s)

X An assignment of the invention to International Business Machines Corporation (includes Recordation Form Cover Sheet)

X A certified copy of a United Kingdom application filed on 30 December 1998, Serial Number 9828750.1

X Information Disclosure Statement, PTO 1449 and copies of references

The fee has been calculated as shown below:

For	Number Filed	Number Extra	Rate		Fee $
Basic Fee					$ 760.00
Total Claims	20 - 20		x 18	=	
Independent Claims	3 - 3		x 78	=	
MULTIPLE DEPENDENT CLAIM PRESENTED			260	=	
				TOTAL	$ 760.00

X Please charge my Deposit Account No. 09-0468 in the amount of $ 760.00. A duplicate copy of this sheet is enclosed.

X The Commissioner is hereby authorized to charge payment of the following fees associated with this communication or credit any overpayment to Deposit Account 09-0468 . A duplicate copy of this sheet is enclosed.

X Any additional filing fees required under 37 CFR §1.16.

X Any patent application processing fees under 37 CFR §1.17.

CERTIFICATE OF MAILING

I hereby certify that the above paper/fee is being deposited with the United States Postal Service as first class mail in an envelope addressed to the Commissioner of Patents and Trademarks, Washington, DC 20231

Date of Deposit June 22, 1999

Person mailing paper/fee Wendy R. Petrovich

Signature _____

Respectfully submitted

Jay P. Sbrollini
Registration No. 36,266
IBM Corp, IP Law Dept,
T J Watson Research Center, PO Box 218,
Yorktown Heights, New York 10598.
Telephone: (914)945- 2587

APPLICATION

FOR

UNITED STATES LETTERS PATENT

DOCKET: POU920030138US1

INTERNATIONAL BUSINESS MACHINES CORPORATION

MEMORY MANAGEMENT

FIELD OF INVENTION

This invention relates to memory management and in particular to registering of object references within a memory management system.

BACKGROUND OF INVENTION

The Java programming language has its origins in a project undertaken by Sun Microsystems to develop a robust programming environment that would meet the technical challenges of the consumer device software environment. The original consumer device projects were eventually abandoned but the Java programming language found itself being used on the World Wide Web to enable cross platform operation of programs downloaded from the internet. It is simple to use having similar features to C++ such as the basic object orientated technology but without some of the more complex features.

Typically, Java applications (source code) are compiled by the Javac compiler into Java byte code (intermediary code or pseudo object code) which can be loaded and executed by a Java Virtual Machine (JVM) (see Figure 1). The JVM provides an instruction set, memory management and program loading capability that is independent of the hardware platform on which it has been implemented. The Java application source code is compiled into architecture independent byte code and the byte code is interpreted by a JVM on the target platform. Java is designed to be portable and follows some defined portability standards, which intend the source code to be

"write once, run anywhere". The Java byte code may be further compiled into machine code (object code) for the target platform at which point the architectural independent nature of Java is lost.

The JVM is a software computing machine, effectively it simulates a hardware machine that processes Java byte code. The byte code is interpreted and processed by a JVM such as an Windows JVM running on a Intel personal computer platform. The JVM includes components for loading class files, interpreting the byte code, garbage collecting redundant objects, and for managing multiple processing threads. The JVM may also include a Just-In-Time compiler to transform some or all the byte code into native machine code.

Garbage collection is the term used for describing how program objects are discarded by the system after they have been loaded into memory and after they are no longer useful. Memory space in object oriented environments is at a premium due to the memory intensive nature of object orientated programs. For further information on garbage collection see Chapter 1 of 'Garbage Collection' by H Jones & R Lins, Wiley. Chapter 4 deals with Mark & Sweep techniques.

Many current implementations of Java use the classic mark-sweep-compact method of garbage collection as delivered in the base SUN JVM. References to the objects that are being processed at any instant by the system are stored in one or more thread stacks and some global variables. The totality of objects that are needed by the system can be found by tracing through the objects referenced in the stacks looking for references to new

objects, tracing the global variables and then tracing through these "root" objects for further references. The objects in use by a system thereby form a graph and any extraneous objects are not part of this graph. Once all the objects in the graph are found, the remaining objects in the heaps may be discarded (garbage collected).

The traditional mark and sweep garbage collection method is described below in terms of pseudo code with respect to a single heap:

- Stop all threads
- Trace all stacks for object references - the local roots
- Trace all classes for object references - the global roots
- Trace through root set for references until no new object references (the sum of the local and global roots is the root set).
- Delete all objects in the single heap that are not referenced.

One of the problems in garbage collecting is tracing a stack for object references when the stack is a mixture of variables including pointers to objects, floating pointer numbers and integer numbers. An accurate scan determines the object pointers exactly whereas a conservative scan determines which words are not object pointers and which maybe. The conservative scan is not exact but it uses less resources than those needed for an accurate scan.

A conservative scan (see Figure 4) retrieves the stack pointer (step 4.1) and then retrieves the word in the stack indicated by the pointer (step 4.2). A first

test is applied, if the word is an object pointer is it pointing in the correct part of the memory, typically this will be between certain limits (step 4.3). If the tested word points outside the limits then it is not an object pointer (4.4). A further test is applied (step 4.5) to check whether the word points to the normal object boundary in the heap? Typically the boundaries in the heap will be multiple number of bytes such as 8 - although there may be several sizes of object grouping say small 8 bytes, medium 64 bytes and large 4096 byte boundary. The word if it is a pointer will point to one of these boundaries and the word is added to the root set step 4.6). If the word does not so point then it is not an object pointer (4.5). If the scan is finished (step 4.7) then the conservative root set has been acquired (4.9). If the scan is not so finished then the stack pointer is incremented and the cycled started again by acquiring the stack word (step 4.2). The root set acquired in this conservative scan is therefore a larger set of words than exist actual object pointers. Furthermore when stack space is created previously existing stack pointers are not immediately overwritten and infiltrate the conservatively scanned root set.

Garbage collection is performed on all the objects in the heap minus the root set and therefore not all the objects that should be are collected. Moreover, since some of the words in the root set are not object pointers, it is fatal to treat them as object pointers for the purposes of updating them when an object is moved. Therefore compaction of objects in a conservatively scanned root set is not desirable.

Accurate scanning has been achieved on the J stack. Techniques exist to find object references in the Java stack which rely on abstract interpretation of the Java code to discover the current stack map at a given set of designated 'safe points'. When execution reaches such a safe point we can do garbage collection in the knowledge that we have a complete map of the where the references are and can update them. A map is stored for each safe point. The map identifies each word in the stack at that point in the process. For a large number of safe points there will be a large number of maps and a high memory usage. A 'safe point' is a point where garbage collection may safely be carried out when there is no object reference both in the stack and in the a register. If this were the case then updating the object reference in the stack during a compaction would not leave a discrepancy between the register pointer and the stack pointer and cause a serious error, possibly a system crash. The Java stack in the JVM holds the variables created and used by the Java application.

The C stack holds the variables created and used by the virtual machine when it interprets the Java application. There is a problem with accurate scanning for garbage collection in the C stack as some object pointers are processed in registers but not placed in the C stack and hence should be in the root set. Most JVM's are compiled from C by a compiler which leaves object pointers in registers as long as possible to improve speed. The disadvantage of this speed optimisation is that many objects are hidden in registers (up to 32) not on the stack and cannot be scanned.

An advantage of the present invention is that the new reference structure forces a C compiler to update the C stack - empty the registers at safe points.

SUMMARY OF INVENTION

According to one aspect of the invention, there is provided a method of retrieving object references in a stack comprising: retrieving an object data structure from the stack; extracting an object reference from one part of the data structure; and extracting a reference to the next data structure in the stack from another part of the data structure.

According to a second aspect there is provided a method of managing a object in a stack based process comprising: storing an object data structure in the stack comprising a reference to the object and a reference to a previously stored object data structure in the stack.

The current JVM does a conservative scan of the C stack to find 32 bit values which may be references to objects. What is required is a method of accurately scanning the C stack. One technique for solving these problems is to indirectly reference objects using handles; however the some JVMs have removed handles to gain a performance improvement. The embodiment describes a solution in which 32 bit quantities are known to be object references by means of some side information. This avoids the problem of integer values masquerading as references causing objects to be needlessly retained. It also allows employment of a compacting collector which is allowed to move objects even if they are referenced from the stacks. Currently the compactor is prevented from

moving such objects leading to reduced compaction efficiency.

There are well known techniques for finding references in the java stack which rely on abstract interpretation of the Java code to discover the current stack map at a given set of designated 'safe-points'. When execution reaches such a safe point garbage collection can be performed in the knowledge of a complete map of where the references are and how they can be updated.

The problem comes with C stacks and locating all local variables which reference objects and which potentially need to updated. Ideally the C compiler would produce stack maps at safe points but this would clearly be a complex organisational task and use large amounts of resource. The advantage of this proposal offers an alternative, very lightweight and compiler independent solution to the problem. All references (typically JHandle* types) are advantageously replaced by a new type, e.g.

```
struct obj_ref {
Jhandle* ref;
struct obj_ref* next;
};
```

The JVM code is searched to find these JHandle* variables and they are turned into struct obj_ref variables. Usages of those variables then need to be modified hence o=&xyz becomes o.ref=&xyz, a simple modification which causes no runtime performance overhead. To make the variable (called o) visible to the outside world (in particular the garbage collector), a

variable "struct obj_ref* head" is defined in the sys_thread structure and assigned a NULL value on thread creation. Thereafter on entry to each function (or inner block) which declares a struct obj_ref (executeJava is a concrete example which defines "o" which would turn into one of these) the struct obj_ref needs to be linked into the list building on "head". Thus the following:

```
save_head = head;          // remember the head on
                           entry so we can restore on
                           exit
original.next = head;      // link the reference to
                           the next in the chain
                           (lower in the C stack, at a
                           higher address).
head = &original;          // this reference is now
                           the head of the chain
```

On exit from the routine the struct obj_ref is unlinked from the chain by restoring the save_head value into head.

If a JHandle* is passed as a parameter to another routine there are two options. A reference to the obj_ref can be passed and then the ref field copied into a local obj_ref which has already been registered in the callee. Alternatively an indirect reference ref can be made from the callee. This latter is similar to introducing a local handle to the object and makes sense when the number of references to the object is small. Therefore it can be decided on a case by case basis which option is the more efficient. As implied by the above on exit from any function continuing a registered obj_ref, the head must be restored to the above saved value.

The key advantage is that an external observer can discover ALL the referenced objects from the C stack by simply starting from head (visible in the sys_thread) and walking to the terminating NULL. At each point on the walk an obj_ref contains a reference to the next obj_ref. In the simplest case the object pointed to by "ref" can be pinned and in a more complex case the object can be moved to a new address and the reference updated to point at the new position.

BRIEF DESCRIPTION OF DRAWINGS

In order to promote a fuller understanding of this and other aspects of the present invention, an embodiment will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a schematic representation of a platform supporting the embodiment of the present invention;

Figure 2 is a schematic representation of a Java Virtual Machine embodying the invention;

Figure 3 is a schematic representation of an active C stack and heap of the JVM of the embodiment;

Figure 4 shows the process of a conservative scan of the C stack according to the prior art; and

Figure 5 shows the process of an accurate scan of the C stack according to the present embodiment.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENT

Referring to Figure 1 there is shown a computer platform 10 typically comprising: a motherboard with a pentium processor, 32Mbytes RAM memory, and interface circuitry; 4G Byte hard drive; 20x CD ROM; SVGA monitor;

keyboard and mouse. On power on the platform loads into the memory an operating system 12 such as Windows NT v4 and subsequently a Java Virtual Machine (JVM) based on the Java Development Kit (JDK) v1.1 or v1.2. The Java virtual machine is capable of loading a Java application 16 from disk into memory so that the application may be executed on the platform. The Java application 16 is object orientated program code and creates Java language objects 18A, 18B, 18C, 18D.

The JVM 14 comprises code components for the multitude of tasks it has to perform (see Figure 2). The components of relevance to the present embodiment comprise memory manager 20 and Java interpreter 22. In operation the JVM uses runtime data area 26, for storing the objects 18 and data for the application 16. The runtime data area 26 includes thread memory space 28 and global heap 34. The thread memory space 28 stores objects and execution data in a local heap (not shown) or a global heap 34.

Execution data including object references and register data are stored in the stacks of which there are two types: the J stack 30 and the C stack 31. The J stack 30 is used to store the Java application execution data when it is being processed by the JVM. The C stack 31 is used to store the JVM execution data when it is being processed by the platform. The JVM is a stack based processor and therefore all the Java application variables are placed in the stack at some point during processing. On the other hand most hardware platforms are register based processors and use a combination of stack and register storage when processing.

The memory manager 20 includes an object allocator 21A and a garbage collector 21B which between them control the creation and deletion of objects within the heap 34.

The Java interpreter 22 includes functionality to interpret (interpreter code 30) and process (object function code 23) the Java byte codes which make up the Java byte code function set. Although only one object function 23 is represented there are many such functions within the JVM.

The object allocator 21A, Interpreter Code 30 and object functions are typical code components in the JVM. Their relevance to the present embodiment is that they define and use object references. In the present embodiment each object reference definition is replaced by a structured object definition 31 and each use of a object reference is replaced by the use of a structured object reference 32. Additionally, each time a structured object is defined it is linked and later unlinked to the C stack 31 by link/unlink code 38. Furthermore to enable safe garbage collection during the operating of the JVM there is interspersed safe point code 34 throughout the object allocator 21A, interpreter code 30 and object functions 23.

A typical object function defines one or more object references and performs operations involving variables including the object references before returning to the calling function. For instance:

```
Void Javafunction (......){
     define object reference O;
     O = &xyz;
     return
     }
```

defines an object reference and sets it to point at the object xyz before returning to the function which called it.

The object function of the embodiment comprises: a structured object reference definition 31 instead of the object reference definition; a structured object reference 32 instead of a object reference; linking and unlinking code 38 for constructing and deconstructing a linked list of structured object references; and safe point code 38 for forcing register variables onto the stack. An example of an object function 23 of the embodiment in pseudo code similar is:

```
Void Javafunction' (......){
     Define structured object reference O; 31
     Link (O); 38
     O.ref=xyz; 32
     if (need_to_be_safe)then call_external_GC(); 36
     unlink (O); 38
     Return
     }
```

The object reference O has been replaced by a new type, a structured object reference O 31 having two fields: the first field being an object reference and the second field being a reference to the next structured

object reference in the stack. An example of pseudo code for Def. struct obj_ref 31 is:

```
Struct obj_ref{
Object reference*ref;
Struct obj_ref*next;
}
```

Link (O) and unlink (O) 38 are instructions to connect and disconnect a structured object reference with the previously stored structured object. An example pseudo code for Link/unlink 38 the takes the form

```
save_head=head ;the variable head points to the
                stack and the variable save_head
                temporally saves this pointer for
                restoration later
O.next=head    ;sets the next field in O to the top
                of the stack
head=&O        ;&O indicates the address of object O
.              ;function is performed
.
.
head=save_head
return
```

The top of the stack is represented by the head variable, this is saved on entry in the save_head variable so that it may be restored on exit. The structured data object has its next field set to point at the top of the stack so that the structured data is linked to the object reference next in the chain (lower in the C stack at a higher address). The head variable is then set to point at the new head of the chain in the C stack. Before exiting from the routine the references

are unlinked from the chain by restoring the save_head value into head.

The use of struct obj_ref 32 is represented by the setting of the next field in the structured data object 'O.next=head' in the previous example. Ordinarily the object reference would not have additional fields and would be set simple by equating it with a variable like 'O=head'.

The safe point code 34 is denoted by the pseudo code 'if (need_to_be_safe) then call_external_GC()'. The flag (need_to_be_safe) is set by one thread when waiting for other threads to finish so that garbage collection may be performed. When a thread encounters a safe point code, it checks the flag and makes the external call to the garbage collector to await the suspension of all other thread processing. Due to the external nature of the call it is necessary to consider the registers that may contain object references are placed in the stack.

Another thread could not simply stop another thread, change any of the refs and then expect that thread to resume safely; clearly there are many points during which there will be copies of object references in registers and those copies WILL be used after the copy on the stack has been changed. The C compiler which compiles the JVM from the C source code is unaware of this problem and will try to put many object references in registers for optimization purposes. Safe points can be forced at any point where the compiler makes an external call since the compiler has no way to predict the operations in the callee and the effect of those operations on the in-memory state. External in the sense that the callee is

not visible at compilation time, ie in another compilation unit. Following such an external call the compiler will force all register copies of externally visible variables to be invalid and will reload from the stack/heap/statics etc. The only time it can avoid such a reload is when it can be absolutely sure that nobody else can have gained addressability to it and is thus unable to change it candidates are clearly variables of purely local scope. Variables of obj_ref type are marked externally visible by virtue of the assignment of their address to an externally visible variable. Hence we can at any point inject a safe point into existing code by something resembling the following:

```
if  (need_to_be_safe)  { // a flag which someone
                           sets to force some GC
                           action
call_external_GC(); // some GC action which wants to
                    possible modify object
                    references and at this point the
                    compiler will reload its 'local'
                    variables
}
```

The big advantage of this is that it should have minimal impact on the performance and will enable exact compacting garbage collection. This in turn will improve the heap management and improve the performance of the JVM.

The garbage Collector 21B comprises accurate scan code 40 and may contain some structured object references 32. An accurate scan is described with reference to Figure 5.

An active C stack 31 of the embodiment comprises a number of active structured object references 32A,B,C and one or more obsolete structured object references 32D (see Figure 3). Each object pointer field of the structured object references 32A,B,C points to respect objects 18A,B,C in Heap 32A. The head variable 44, part of thread data, points to the structured object reference at the top of the C stack 31. The linked list comprises structured object reference 32C at the top, 32A in the middle and 32B at the bottom. The reference pointer field (next field) of structured object reference 32C points to structured object reference 32A. The reference pointer field (next field) of structured object reference 32A points to structured object reference 32B. The reference pointer field (next field) of structured object reference 32B contains a null value indicating the end of the linked list.

The conservative scan depicted in Figure 4 has been described in the introduction to the specification.

An accurate scan of the C stack of the embodiment initially retrieves the head variable 44 which points at the structured object reference at the top of the stack (step 5.1) - this is made the current structured reference object (see Figure 5). The object reference of the current structured object reference is acquired and added to the root set - step 5.2. The reference of the next structured object reference is acquired from the current structured object reference - step 5.3 and is examined for a null value which would indicate the last object reference in the list - step 5.4. If there is no null value the current structured object reference is

updated to this next structured object reference - step 5.5 and the next object reference is added to the root set at step 5.2 where the process starts again. If there is a null value the accurate scan terminates with a complete root set of object pointers - step 5.6.

In summary there is described a method of accurately scanning the C stack. One technique for solving these problems is to indirectly reference objects using handles however the some JVMs have removed handles to gain a performance improvement. The embodiment describes a solution in which 32 bit quantities are known to be object references by means of some side information. This avoids the problem of integer values masquerading as references causing objects to be needlessly retained. It also allows employment of a compacting collector which is allowed to move objects even if they are referenced from the stacks. Currently the compactor is prevented from moving such objects leading to reduced compaction efficiency.

("Java" is a trademark of Sun Microsystems Inc.)

Now that the invention has been described by way of a preferred embodiment, various modifications and improvements will occur to those person skilled in the art. Therefore it should be understood that the preferred embodiment has been provided as an example and not as a limitation.

CLAIMS

1. A method of retrieving object references in a stack comprising:

retrieving an object data structure from the stack;

extracting an object reference from one part of the data structure; and

extracting a reference to the next data structure in the stack from another part of the data structure.

2. A method as claimed in claim 1 further comprising retrieving the next data structure and retrieving the corresponding object reference and corresponding next data structure reference.

3. A method as claimed in claim 2 further comprising retrieving all linked data structures in the stack.

4. A method as claimed in claim 3 further comprising retrieving a last data structure having no next data structure reference.

5. A method as claimed in claim 4 further comprising retrieving the first object data structure in the stack referenced by a first object data structure pointer.

6. A method as claimed in claim 5 further comprising using the retrieved object references to define a root set of objects.

7. A method as claimed in claim 6 comprising:

defining a reachable set of objects as all objects referenced directly or indirectly by the root set objects.

8. A method as claimed in claim 7 further comprising identifying all objects within the process and reclaiming the memory space of all non-reachable objects.

9. A method as claimed in claim 7 or 8 further comprising moving reachable objects so that they are contiguous in memory and updating all object references in the stack by tracing through the chain of object data structures.

11. A method of managing a object in a stack based process comprising:

storing an object data structure in the stack comprising a reference to the object and a reference to a previously stored object data structure in the stack.

12. A method as claimed in claim 11 further comprising linking the object data structure to the previously stored object data structure.

13. A method as claimed in claim 12 further comprising:

storing a variable pointing to the previously stored object data structure at the top of the stack;

using the variable when storing a new object data structure; and

updating the variable with the new object data structure reference.

14. A method as claimed in claim 13 further comprising:
saving the variable pointer;
storing the object data structure;
updating the variable with the reference to the latest stored object data structure;
performing the process ; and
restoring the stack pointer.

15. A method as claimed in claim 11 further comprising:
retrieving an object data structure and extracting the associated object reference and data structure reference;
using the associated data structure reference to retrieve the previously stored object data structure;
retrieving all the object references in the stack by tracing through the chained of object data structures.

16. A method as claimed in claim 15 further comprising using the retrieved object pointers to identify a root set of objects.

17. A method as claimed in claim 16 comprising:
identifying all objects referenced directly or indirectly by the root set objects and marking the root set and all referenced objects as reachable.

18. A method as claimed in claim 17 further comprising identifying all objects within the process and reclaiming the memory space of all non-reachable objects.

19. A method as claimed in claim 18 further comprising moving reachable objects in process memory so that they are contiguous and updating all object references in the stack by tracing through the chain of object data structures.

20. A system for retrieving object references in a stack comprising:

means for retrieving an object data structure from the stack;

means for extracting an object reference from one part of the data structure; and

means for extracting a reference to the next data structure in the stack from another part of the data structure.

ABSTRACT

MEMORY MANAGEMENT

There is described a method of accurately scanning the C stack. One technique for solving these problems is to indirectly reference objects using handles however the some JVMs have removed handles to gain a performance improvement. The embodiment describes a solution in which 32 bit quantities are known to be object references by means of some side information. This avoids the problem of integer values masquerading as references causing objects to be needlessly retained. It also allows employment of a compacting collector which is allowed to move objects even if they are referenced from the stacks. Currently the compactor is prevented from moving such objects leading to reduced compaction efficiency.

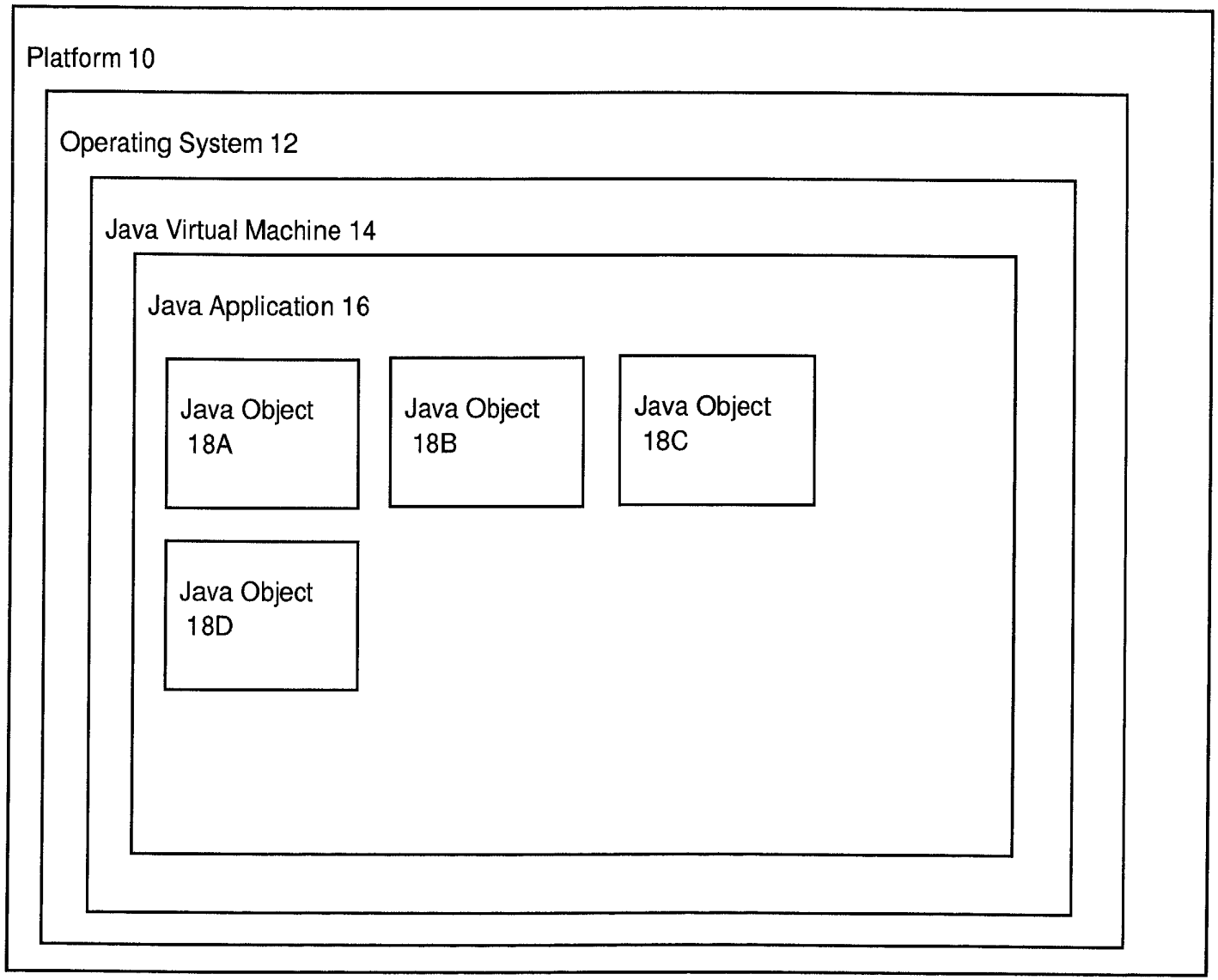
Platform 10
Operating System 12
Java Virtual Machine 14
Java Application 16
Java Object 18A
Java Object 18B
Java Object 18C
Java Object 18D

Figure 1

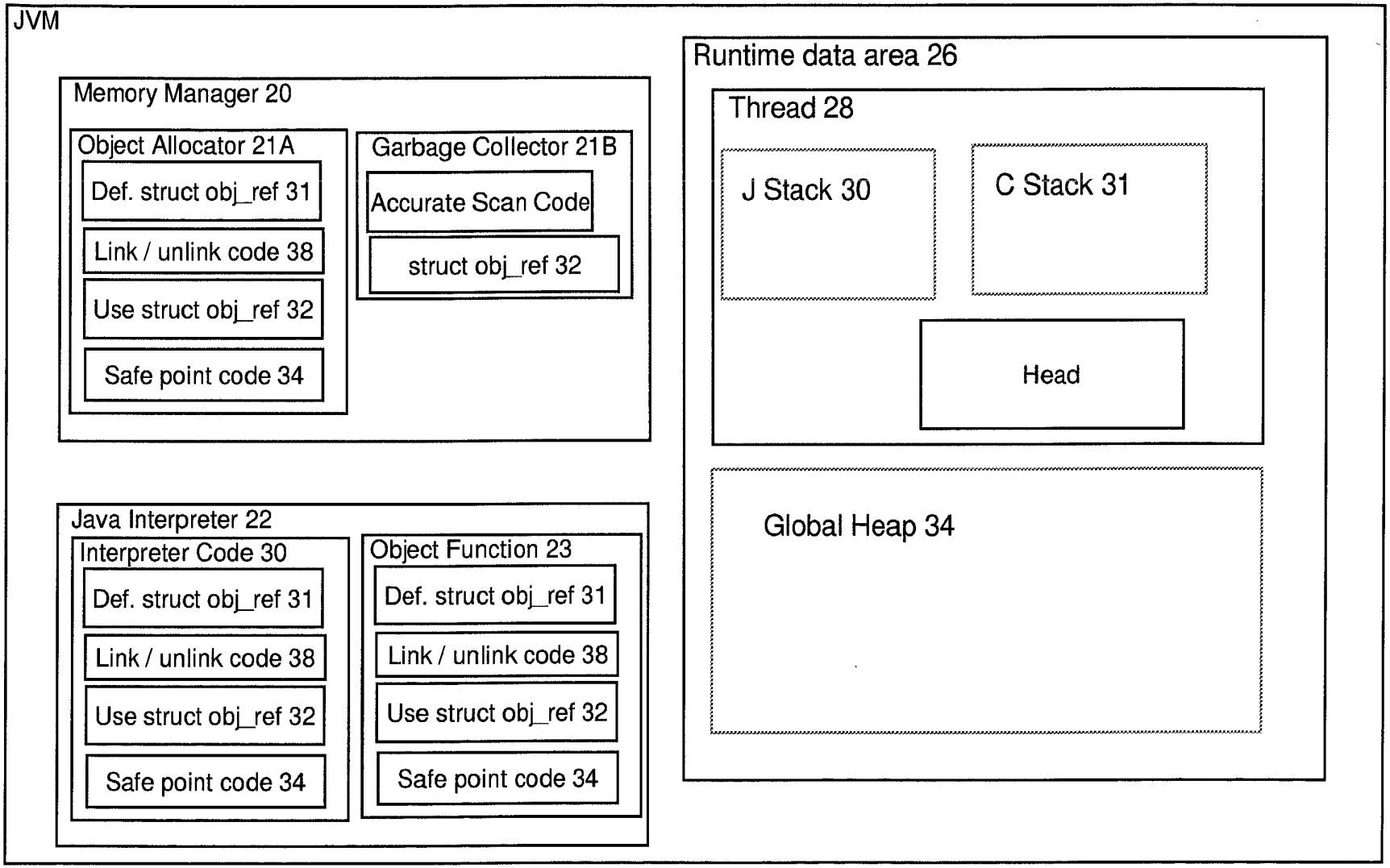
JVM
Memory Manager 20
Object Allocator 21A
Def. struct obj_ref 31
Link / unlink code 38
Use struct obj_ref 32
Safe point code 34
Garbage Collector 21B
Accurate Scan Code
struct obj_ref 32
Java Interpreter 22
Interpreter Code 30
Def. struct obj_ref 31
Link / unlink code 38
Use struct obj_ref 32
Safe point code 34
Object Function 23
Def. struct obj_ref 31
Link / unlink code 38
Use struct obj_ref 32
Safe point code 34
Runtime data area 26
Thread 28
J Stack 30
C Stack 31
Head
Global Heap 34

Figure 2

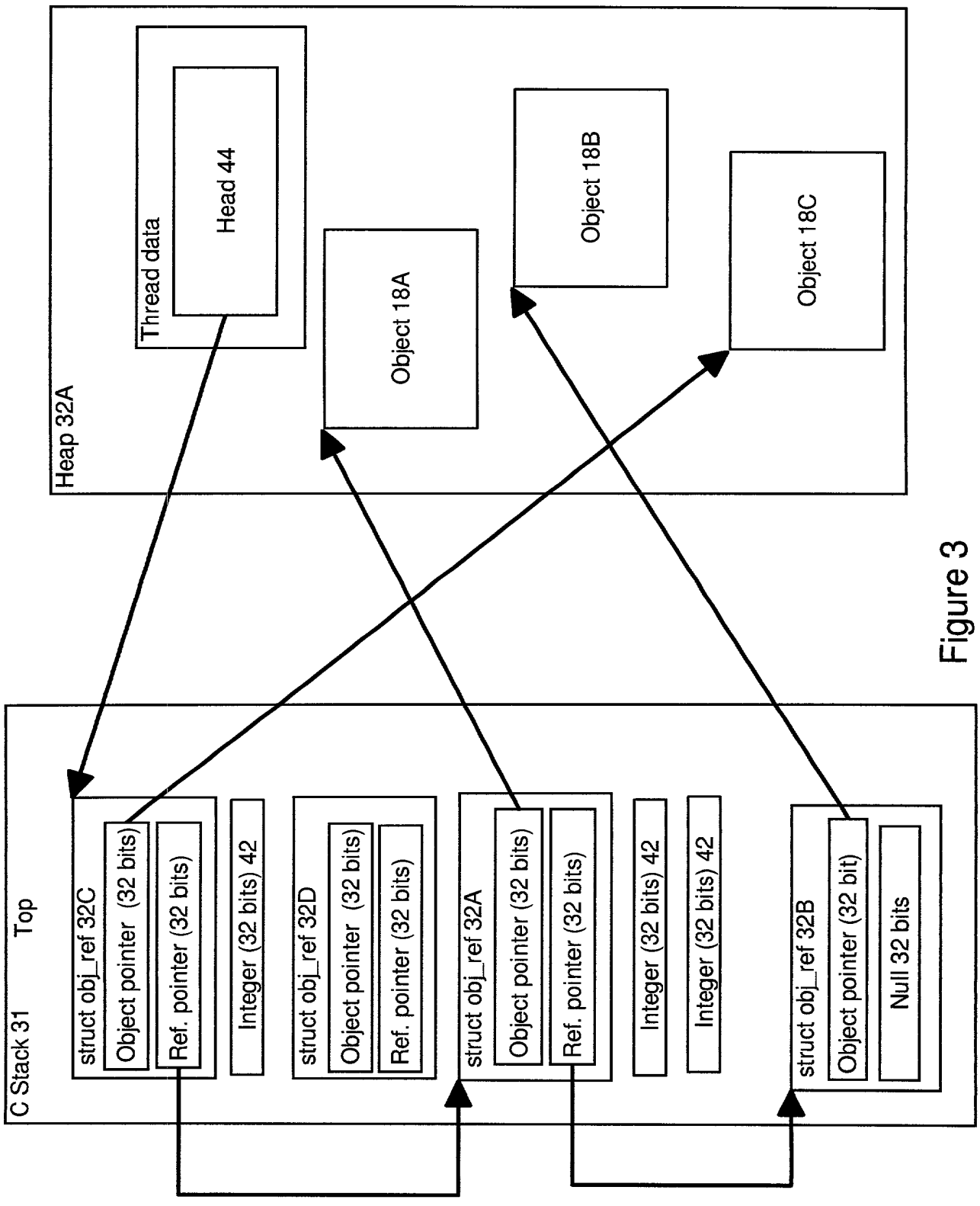
C Stack 31
Top
struct obj_ref 32C
Object pointer (32 bits)
Ref. pointer (32 bits)
Integer (32 bits) 42
struct obj_ref 32D
Object pointer (32 bits)
Ref. pointer (32 bits)
struct obj_ref 32A
Object pointer (32 bits)
Ref. pointer (32 bits)
Integer (32 bits) 42
Integer (32 bits) 42
struct obj_ref 32B
Object pointer (32 bit)
Null 32 bits
Heap 32A
Thread data
Head 44
Object 18A
Object 18B
Object 18C

Figure 3

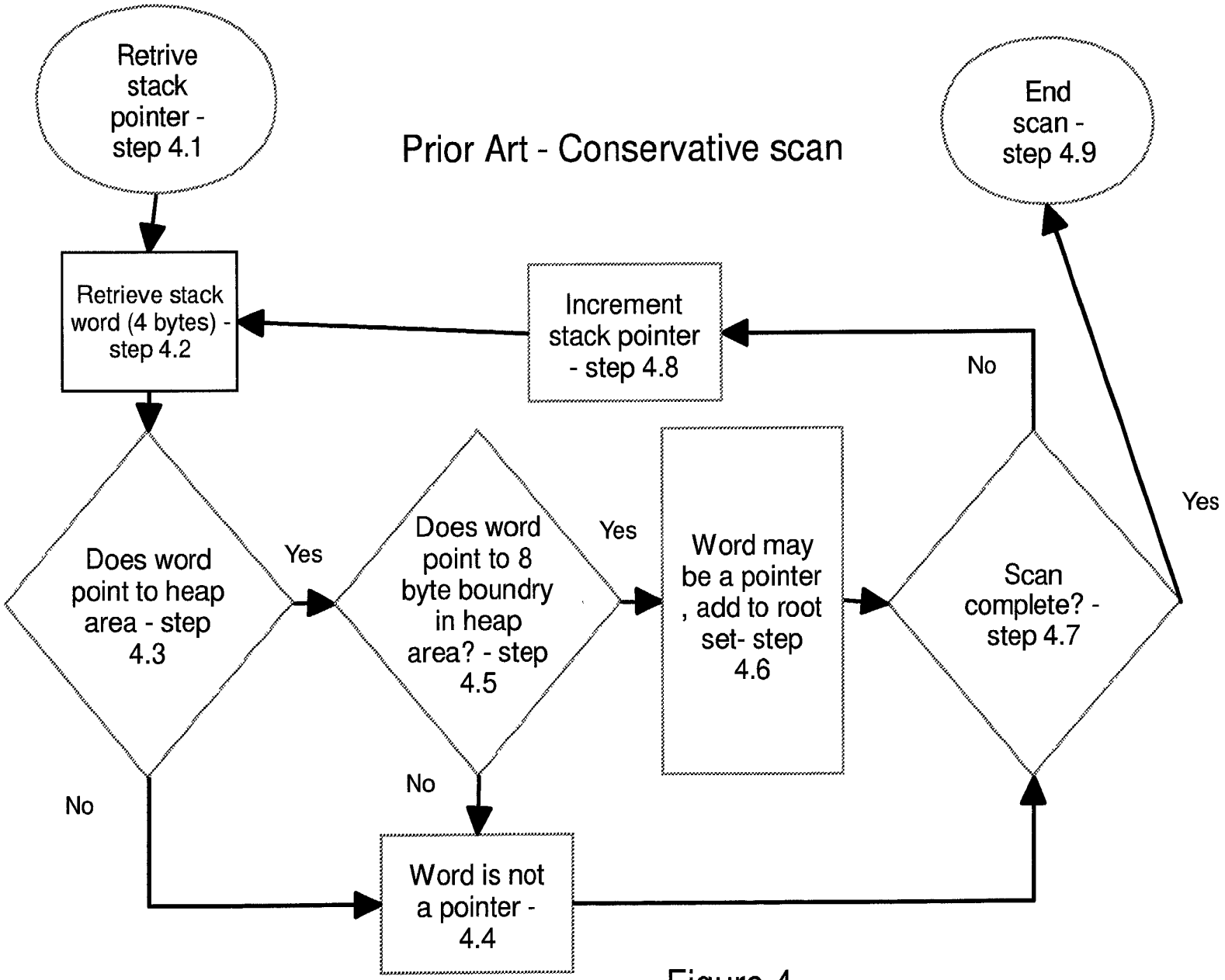
Retrive stack pointer - step 4.1
Prior Art - Conservative scan
End scan - step 4.9
Retrieve stack word (4 bytes) - step 4.2
Increment stack pointer - step 4.8
No
Does word point to heap area - step 4.3
Yes
Does word point to 8 byte boundry in heap area? - step 4.5
Yes
Word may be a pointer , add to root set- step 4.6
Scan complete? - step 4.7
Yes
No
No
Word is not a pointer - 4.4

Figure 4

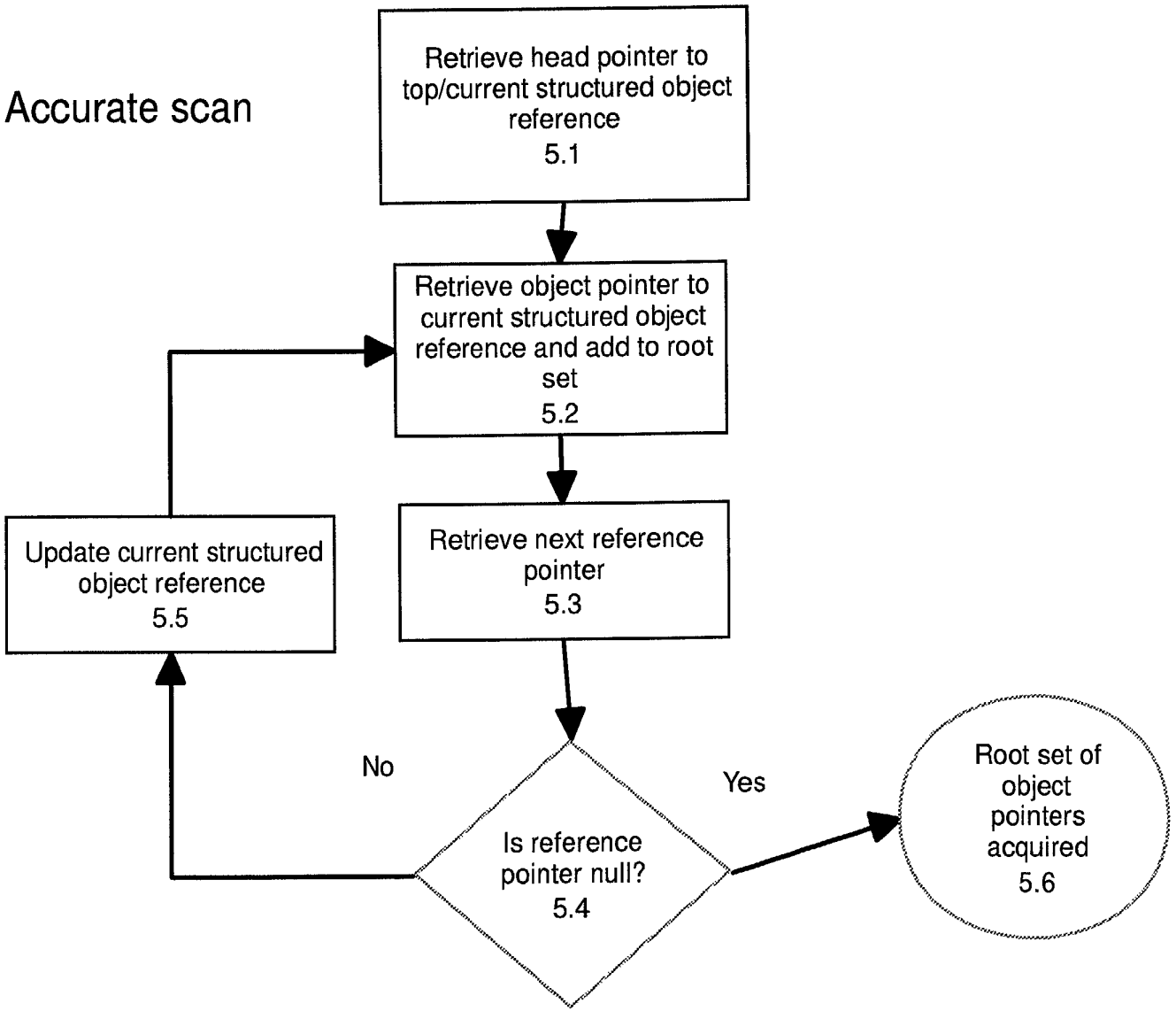
Accurate scan
Retrieve head pointer to top/current structured object reference
5.1
Retrieve object pointer to current structured object reference and add to root set
5.2
Retrieve next reference pointer
5.3
Is reference pointer null?
5.4
No
Yes
Update current structured object reference
5.5
Root set of object pointers acquired
5.6

Figure 5

DOCKET NO: UK998092

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name; I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled

MEMORY MANAGEMENT

the specification of which (check one)

__X__ is attached hereto.

_____ was filed on
as Application Serial No.
and was amended on

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with Title 37, CFR 1.56.

I hereby claim foreign priority benefits under Title 35, USC 119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s):

9828750.1 Number	United Kingdom Country	30 December 1998 Filing Date Day/Month/Year	__X__Yes _____No Priority Claimed

I hereby claim the benefit under Title 35, USC 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, USC 112, I acknowledge the duty to disclose information material to the patentability of this application as defined in Title 37, CFR 1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

Prior U.S. Applications:

Serial No.	Filing Date	Status

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

POWER OF ATTORNEY: As a named inventor, I hereby appoint the following attorneys and/or agents to prosecute this application and transact all business in the Patent and Trademark Office connected therewith:

Manny W. Schecter, Reg. No. 31,722; William B. Porter, Reg. No. 33,135; Douglas W. Cameron, Reg. No. 31,596; Kevin M. Jordan, Reg. No. 40,277; Stephen C. Kaufman, Reg. No. 29,551; Richard M. Ludwin, Reg. No. 33,010; Daniel P. Morris, Reg. No, 32,053; Louis J. Percello, Reg. No. 33,206; Jay P. Sbrollini, Reg. No. 36,266; Stephen S. Strunck, Reg. No. 28,672; Robert P. Tassinari, Reg. No. 36,030; Robert M. Trepp, Reg. No. 25,933; Marc D. Schechter, Reg. No. 28,989; Louis P. Herzberg, Reg. No. 41,500.

Send all correspondence to:

Jay P. Sbrollini
IBM Corp, IP Law Dept,
T J Watson Research Center, PO Box 218,
Yorktown Heights, New York 10598.
Telephone: (914)945- 2587

FULL NAME OF SOLE OR FIRST INVENTOR: **Martin John Trotter**

INVENTOR'S SIGNATURE: Martin John Trotter DATE: 13th May 9999

RESIDENCE: 2 Broadgate Cottages, Potters Heron Close, Ampfield, Romsey, Hampshire SO51 9BX, United Kingdom

CITIZENSHIP: British

POST OFFICE ADDRESS: As above

FULL NAME OF SECOND INVENTOR:

INVENTOR'S SIGNATURE: _____ DATE: _____

RESIDENCE:

CITIZENSHIP:

POST OFFICE ADDRESS:

FULL NAME OF THIRD INVENTOR:

INVENTOR'S SIGNATURE: _____ DATE: _____

RESIDENCE:

CITIZENSHIP:

POST OFFICE ADDRESS: